APPENDIX C

Backtracking

Backtracking is a general algorithm that finds all or some solutions to any computational problem that incrementally builds candidates to the solutions. At each step, while the valid candidates to the solution are extended, the other candidates are discarded. That is, each partial candidate *c* is immediately abandoned after it is determined that *c* cannot be a possible and valid solution to the problem. Hence, the name, backtrack.

The concept of backtracking is applicable only to problems that support partial candidate solutions and a relatively quick test to determine if the partial candidate solution can be completed to a valid solution. Backtracking is extensively used to solve constraint satisfaction problems such as following:

- Puzzles like Sudoku, eight queen's problem, crosswords, verbal arithmetic, Solitaire
- Combinational optimization problem like parsing and Knapsack problem
- Logic programming languages that internally use backtracking include Icon, Planner and Prolog
- *diff* which is a version comparing engine for the MediaWiki software

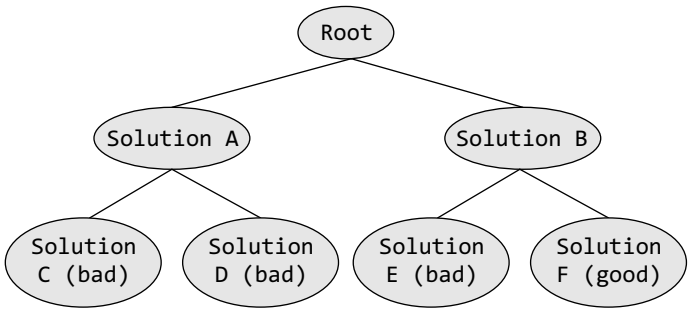
Backtracking is said to be a meta-heuristic algorithm (in contrast to a specific algorithm) that is guaranteed to find all solutions to a finite problem in a bounded amount of time. The term meta-heuristic implies that the algorithm works based on user-given procedures that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates.

Conceptualizing the backtracking process as potential search tree, the partial candidates of the solution can be viewed as the nodes of the tree. Each partial candidate has child nodes that represent the candidates that differ from it by a single extension step. Of course, the leaf nodes are the partial candidates that cannot be extended further.

The backtracking algorithm recursively traverses the search tree in depth-first order (starting from the root node). At each node *c*, the algorithm checks if *c* can be completed to a valid solution. If it cannot be completed, then the entire sub-tree rooted at *c* is pruned. However, if there is a possibility that *c* may lead to valid solution then the algorithm first checks if *c* itself is a valid solution. If yes, then the algorithm declares *c* as the valid solution otherwise it recursively enumerates all sub-trees of *c*.

Note The tests to determine the valid solution and children of each node are all defined by user-given procedures.

Look at the search tree given below.



Step 1 Start with the root node. The two available options are—Solution A and Solution B. Select Solution A.

Step 2 At Solution A there are again two options—Solution C and Solution D. Select Solution C.

Step 3 Since Solution C is not a good (valid and possible) candidate, backtrack to Solution A and now select Solution D.

Step 4 Since Solution D is not a good (valid and possible) candidate, backtrack to Solution A. now there are no more options available at Solution A, so select Solution B.

Step 5 At Solution B there are two options—Solution E and Solution F. Select Solution E.

Step 6 Since Solution E is not a good (valid and possible) candidate, backtrack to Solution B and now select Solution F.

Step 7 Solution F is a good (valid and possible) candidate, so the algorithm stops here.

Advantage: By terminating searches and not exploring candidates that cannot lead to valid possible solutions, the backtracking technique reduces the number of nodes examined. The algorithm can be used to solve exponential time problems in a *reasonable* amount of time.

Note Backtracking is not an optimization technique. It just reduces the search space.